

SE117, Spring 2016
Lab 1: Fun with Collections
Assigned: April 21
Due: May 12 23:59:59

TA: Weilun Xiong (azardf4yy@gmail.com)

Introduction

In your role as a client of these collections classes with the low-level details abstracted away, you can put your energy toward solving more interesting problems. In this assignment, your job is to write two short client programs that use these classes to do nifty things. The tasks may sound a little daunting at first, but given the power tools in your arsenal, each requires only a page or two of code. Let's hear it for abstraction!

Problem I: Word ladders

Introduction

A **word ladder** is a connection from one word to another formed by changing one letter at a time with the constraint that at each step the sequence of letters still forms a **valid** word.

For example, here is a word ladder connecting "code" to "data"

code -> core -> care -> dare -> date -> data

That word ladder, however, is not the shortest possible one. Although the words may be a little less familiar, the following ladder is one step shorter:

code -> cade -> cate -> date -> data

Requirements

Your job in this problem is to write a program that finds a minimal *word ladder* between two words. Your code will make use of several of C++ standard library, along with breadth-first search algorithm to find the shortest such sequence.

You may try out the demo program before you implement it. Your word ladder may be different from our demo, because there may be more than one shortest word ladder. You can test whether a word is a valid English word by using `Lexicon` and `EnglishWords.dat` that we prepared.

Here, for example, is a sample run of the `word-ladder` program in operation:

```
Enter start word (RETURN to quit): hello
Enter destination word: world
Found ladder: hello hells heals weals weald woald world
Enter start word (RETURN to quit):
```

Word Ladder Implementation

You can ignore this section if you have already known what breadth-first search is.

Finding a word ladder is a specific instance of a shortest-path problem, in which the challenge is to find the shortest path from a starting position to a goal. Shortest-path problems come up in a variety of situations such as routing packets in the Internet, robot motion planning, determining proximity in social networks, comparing gene mutations, and more.

One strategy for finding a shortest path is the classic algorithm known as breadth-first search, which is a search process that expands outward from the starting position, considering first all possible solutions that are one step away from the start, then all possible solutions that are two steps away, and so on, until an actual solution is found. Because you check all the paths of length 1 before you check any of length 2, the first successful path you encounter must be as short as any other.

For word ladders, the breadth-first strategy starts by examining those ladders that are one step away from the original word, which means that only one letter has been changed. If any of these single-step changes reach the destination word, you're done. If not, you can then move on to check all ladders that are two steps away from the original, which means that two letters have been changed. In computer science, each step in such a process is called a *hop*.

The breadth-first algorithm is typically implemented by using a queue to store partial ladders that represent possibilities to explore. The ladders are enqueued in order of increasing length. The first elements enqueued are all the one-hop ladders, followed by the two-hop ladders, and so on. Because queues guarantee first-in/first-out processing, these partial word ladders will be dequeued in order of increasing length.

To get the process started, you simply add a ladder consisting of only the start word to the queue. From then on, the algorithm operates by dequeuing the ladder from the front of the queue and determining whether it ends at the goal. If it does, you have a complete ladder, which must be minimal. If not, you take that partial ladder and extend it to reach words that are one additional hop away, and enqueue those extended ladders, where they will be examined later. If you exhaust the queue of possibilities without having found a completed ladder, you can conclude that no ladder exists.

Another issue that is a bit subtle is the restriction that you not reuse words that have been included in a previous ladder. One advantage of making this check is that doing so reduces the need to explore redundant paths. If there is a word ladder that begins with `cat -> cot -> con -> cog`, then

there must be a shorter one that, in effect, cuts out the middleman by eliminating the unnecessary word con. In fact, as soon as you've enqueued a ladder ending with a specific word, you never have to enqueue that word again.

The simplest way to implement this strategy is to keep track of the words that have been used in any ladder (which you can easily do using another lexicon) and ignore those words when they come up again. Keeping track of what words you've used also eliminates the possibility of getting trapped in an infinite loop by building a circular ladder.

One of the other questions you will need to resolve is what data structure you should use to represent word ladders. Conceptually, each ladder is just an ordered list of words, which should make your mind scream out "Vector!" (Given that all the growth is at one end, stacks are also a possibility, but vectors will be more convenient when you are trying to print out the results.) The individual components of the Vector are of type string.

It is possible to make the algorithm considerably more concrete by implementing it in pseudo code, which is simply a combination of actual code and English. The pseudo code for the word-ladder problem:

```
Create an empty queue.
Add the start word to a new ladder.
Add the ladder to the queue.
while (the queue is not empty) {
    Dequeue the first ladder from the queue.
    if (the final word in this ladder is the destination word) {
        Return this ladder as the solution.
    }
    for (each word in the lexicon of English words that differs by one letter) {
        if (that word has not already been used in a ladder) {
            Create a copy of the current ladder.
            Add the new word to the end of the copy.
            Add the new ladder to the end of the queue.
        }
    }
}
Report that no word ladder exists.
```

Problem II: Random Writer

Introduction

In the past few decades, computers have revolutionized student life. In addition to providing no end of entertainment and distractions, computers also have also facilitated much productive student work. However, one important area of student labor that has been painfully neglected is the task of filling up space in papers, Ph.D. dissertations, grant proposals, and recommendation letters with important sounding and somewhat sensible random sequences. Come on, you know the overworked

TA/professor/reviewer doesn't have time to read too closely.

To address this burning need, the *random writer* is designed to produce somewhat sensible output by generalizing from patterns found in the input text. When you're coming up short on that 10-page paper due tomorrow, feed in the eight pages you already have written into the random writer, and "Voila!" another couple of pages appear. You can even feed your own .cpp files back into your program and have it build a new random program on...

Requirements

Your program is to read a source text, build an order- k Markov model for it, and generate random output that follows the frequency patterns of the model.

First, you prompt the user for the name of a file to read for the source text and reprompt as needed until you get a valid name. (And you probably have a function like this lying around somewhere.) Now ask the user for what order of Markov model to use (a number from 1 to 10). This will control what seed length you are working with.

Use simple character-by-character reading on the file. As you go, track the current seed and observe what follows it. Your goal is to record the frequency information in such a way that it will be easy to generate random text later without any complicated manipulations.

Once the reading is done, your program should output **2000 characters** of random text generated from the model. For the initial seed, choose the sequence that appears **most frequently** in the source (e.g., if you are doing an order-4 analysis, the four-character sequence that is most often repeated in the source is used to start the random writing). If there are several sequences tied for most frequent, any of them can be used as the initial seed. Output the initial seed, then **choose the next character based on the probabilities of what followed that seed in the source**. Output that character, update the seed, and the process repeats until you have 2000 characters.

For example, consider an order-2 Markov model built from this sentence from Franz Kafka's *Metamorphosis*:

As Gregor Samsa awoke one morning from uneasy dreams he found himself transformed in his bed into a gigantic insect.

Here is how the first few characters might be chosen:

- ♦ The most commonly occurring sequence is the string "in", which appears four times. This string therefore becomes the initial seed.
- ♦ The next character is chosen based on the probability that it follows the seed "in" in the source. The source contains four occurrences of "in", one followed by g, one followed by t, one followed by s, and one followed by a space. Thus, there should be a 1/4 chance each of choosing g, t, s, or space. Suppose space is chosen this time.
- ♦ The seed is updated to "n ". The source contains one occurrence of "n ", which is followed by h. Thus the next character chosen is h.
- ♦ The seed is now " h". The source contains three occurrences of " h", once followed by e, and twice followed by i. Thus, there is a 1/3 chance of choosing e and 2/3 for i. Imagine i is

chosen this time.

- ♦ The seed is now "hi". The source contains two occurrences of "hi", once followed by m, the other by s. For the next character, there is 1/2 chance of choosing m and 1/2 chance for s.

If your program ever gets into a situation in which there are **no characters to choose** from (which can happen if the only occurrence of the current seed is at the exact end of the source), your program can just **stop writing early**.

Details of Markov model

You can ignore this section if you have already known what Markov model is.

Random writing is based on an idea advanced by Claude Shannon in 1948 and subsequently popularized by A. K. Dewdney in his *Scientific American* column in 1989. Shannon's famous paper introduces the idea of a Markov model for English text. A **Markov model** is a statistical model that describes the future state of a system based on the current state and the conditional probabilities of the possible transitions. Markov models have a wide variety of uses, including recognition systems for handwriting and speech, machine learning, and bioinformatics. Even Google's PageRank algorithm has a Markov component to it. In the case of English text, the Markov model is used to describe the possibility of a particular character appearing given the sequence of characters seen so far. The sequence of characters within a body of text is clearly not just a random rearrangement of letters, and the Markov model provides a way to discover the underlying patterns and, in this case, to use those patterns to generate new text that fits the model.

First, consider generating text in total randomness. Suppose you have a monkey at the keyboard who is just as likely to hit any key as another. While it is theoretically possible - given enough monkeys, typewriters, and ages of the universe - that this sort of random typing would produce a work of Shakespeare, most output will be gibberish that makes pretty unconvincing English:

No model aowk fh4.s8an zp[q;lk ss4o2mai/

A simple improvement is to gather information on character frequency and use that as a weight for choosing the next letter. In English text, not all characters occur equally often. Better random text would mimic the expected character distribution. Read some input text (such as Mark Twain's *Tom Sawyer*, for example) and count the character frequencies. You'll find that spaces are the most common, that the character e is fairly common, and that the character q is rather uncommon. Suppose that space characters represent 16% of all characters in *Tom Sawyer*, e just 9%, and q a mere .04% of the total. Using these weights, you could produce random text that exhibited these same probabilities. It wouldn't have a lot in common with the real *Tom Sawyer*, but at least the characters would tend to occur in the proper proportions. In fact, here's an example of what you might produce:

Order 0

rla bsht eS ststof hhfosdsdewno oe wee h .mr ae
iriii ela iad o r te u t mnyto onmalysnce, ifu en
c fDwn oee iteo

This is an *order-0 Markov model*, which predicts that each character occurs with a fixed probability, independent of previous characters. We're getting somewhere, but most events occur in context. Imagine randomly generating a year's worth of Fahrenheit temperature data. A series of 365 random

integers between 0 and 100 wouldn't fool the average observer. It would be more credible to make today's temperature a random function of yesterday's temperature. If it is 85 degrees today, it is unlikely to be 15 degrees tomorrow. The same is true of English words: If this letter is a *q*, then the following letter is quite likely to be a *u*. You could generate more realistic random text by choosing each character from the ones likely to follow its predecessor. For this, process the input and build an order-1 model that determines the probability with which each character follows another character. It turns out that *s* is much more likely to be followed by *t* than *y* and that *q* is almost always followed by *u*. You could now produce some randomly generated *Tom Sawyer* by picking a starting character and then choosing the character to follow according to the probabilities of what characters followed in the source text. Here's some output produced from an *order-1 model*:

Order 1

"Shand tucthiney m?" le ollds mind Theybooure He, he swhit Pereg lenigabo Jodind alllld ashanthe ainofevids trelin--p asto oun theanthadomoere
--

This idea extends to longer sequences of characters. An order-2 model generates each character as a function of the two-character sequence preceding it. In English, the sequence *sh* is typically followed by the vowels, less frequently by *r* and *w*, and rarely by other letters. An order-5 analysis of *Tom Sawyer* reveals that *leave* is often followed by *s* or *space* but never *j* or *q*, and that *Sawye* is always followed by *r*. Using an order-*k* model, you generate random output by choosing the next character based on the probabilities of what followed the previous *k* characters in the input text. This string of characters preceding the current point is called the seed.

At only a moderate level of analysis (say, orders 5 to 7), the randomly generated text begins to take on many of the characteristics of the source text. It probably won't make complete sense, but you'll be able to tell that it was derived from *Tom Sawyer* as opposed to, say, *Pride and Prejudice*. At even higher levels, the generated words tend to be valid and the sentences start to scan. Here are some more examples:

Order 2

"Yess been." for gothin, Tome oso; ing, in to weliss of an'te cle -- armit. Papper a comeasione, and smomenty, fropeck hinticer, sid, a was Tom, be suck tied. He sis tred a youck to themen

Order 4

en themself, Mr. Welshman, but him awoke, the balmy shore. I'll give him that he couple overy because in the slated snufflindeed structure's kind was rath. She said that the wound the door a fever eyes that WITH him.
--

Order 6

Come -- didn't stand it better judgment; His hands and bury it again, tramped herself! She'd never would be. He found her spite of anything the one was a prime feature sunset, and hit upon that of the forever.

Order 8

look-a-here -- I told you before, Joe. I've heard a pin drop. The stillness was complete, how- ever, this is awful crime, beyond the village was sufficient. He would be a good enough to get that

Order 10

night, Tom and Becky.
you understanding that they don't come around in the cave should get the word "beauteous" was over-fondled, and that together" and decided that he might as we used to do -- it's nobby fun. I'll learn you."

Hints

Although it may sound daunting at first glance, this task is supremely manageable with the bag of power tools you bring to the job site.

Map and Vector are just what you need to store the model information. The keys into the map are the possible seeds (e.g., if the order is 2, each key is a 2-character sequence found in the source text). The associated value is a vector of all the characters that follow that seed in the source text. That vector can - and likely will - contain a lot of duplicate entries. Duplicates represent higher probability transitions from this Markov state. Explicitly storing duplicates is the easiest strategy and makes it simple to choose a random character from the correct frequency distribution. A more space-efficient strategy would store each character at most once, with its frequency count. However, it's a bit more awkward to code this way. You are welcome to do either, but if you choose the latter, please take extra care to keep the code clean.

Determining which seed(s) occurs most frequently in the source can be done by iterating over the entries once you have finished the analysis.

To read a file one character at a time, you can use the `get` member function for `ifstream` like this:

```
char ch;
while (input.get(ch)) {
    /* ... process ch ... */
}
```

This will read the file one character at a time, executing the loop until no more characters can be read.

Possible extensions

There are many extensions you could think about introducing into the random writer program—most of which will also make it easier for you to generate a contest entry. One possibility is to extend the Markov model so that the individual units are words rather than characters. Another is to write a program that runs the Markov model in the opposite direction, in which the goal is to recognize an author by his or her characteristic patterns. This requires building multiple Markov models, one for each candidate author, and comparing them to an unattributed text to find the best match. This sort of "literary forensic analysis" has been used to try to determine the correct attribution for texts where the authorship is unknown or disputed.

References

1. Dewdney A. A potpourri of programmed prose and prosody [J]. Scientific American, 1989.

2. Shannon C E. A mathematical theory of communication [J]. ACM SIGMOBILE Mobile Computing and Communications Review, 2001, 5(1): 3-55.
3. Markov chain [EB/OL]. [2013-09-14]. http://en.wikipedia.org/wiki/Markov_chain
4. Free ebooks - Project Gutenberg [G/OL]. [2013-09-14]. <http://www.gutenberg.org>

Hand in

You only need to turn in your `WordLadder.cpp` and `RandomWriter.cpp` files, zip source code into *lab1_{Student Number}.zip*, such as *lab1_123456.zip*, commit to course homework system.